FAST RTPS V1.0.4

USER MANUAL

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 16 – Bajo K
28760 Tres Cantos Madrid
Tel: + 34 91 804 34 48
info@eProsima.com – www.eProsima.com

Trademarks

eProsima is a trademark of *Proyectos y Sistemas de Mantenimiento SL.* All other trademarks used in this document are the property of their respective owners.

License

eProsima Fast RTPS is licensed under the terms described in the FASTRTPS_LIBRARY_LICENSE.txt file included in this distribution.

Technical Support

Phone: +34 91 804 34 48

Email: Support@eProsima.com

Contents

1. Introduction....1
1.1. Purpose....1
1.2. Scope....1
1.3. Intended Audience....1
1.4. Reference Material....1
1.5. Document organization....2
2. Implementation overview: scope and limitations....3
2.1. Implementation scope....3
2.2. Implementation limitations....3
2.3. Implementation Dependencies....3
3. Implementation Specification....4
3.1. System Architecture....4
3.2. Behavior Implementation....5
3.2.1. Thread structure....5
3.2.2. Resource structure....5
3.2.3. Main events....5
4. Object Specification....7
4.1. Data Design....10
4.1.1. Internal Data Structures....10
4.1.2. Global Data Structures....10
4.1.3. Temporary Data Structures....10
4.2. Common Data Types....10
5. Publisher – Subscriber Interface Layer....13
5.1. Domain....13
5.2. Participant....13
5.3. Type definition and registration....14
5.4. Publisher....16
5.4.1. Publisher-based class....16
5.5. Subscriber....18
5.5.1. Subscriber creation....18
5.6. QoS Examples....19
5.6.1. Time-Based Filter and Content-Based Filter Qos....20
5.6.2. Ownership Qos....21
6. RTPS User interface....22

6.1. RTPSDomain....22
6.2. RTPSParticipant....22
6.2.1. Endpoint registration....22
6.3. RTPSWriter – WriterHistory....23
6.3.1. WriterAttributes....23
6.3.2. Send Data with a RTPSWriter....23
6.3.3. Matched readers....24
6.4. RTPSReader – ReaderHistory....24
6.4.1. ReaderAttributes....24
6.4.2. Reading data from a RTPSReader....24
6.4.3. Receving data with a ReaderListener....25
7. Automatic code generation....26
7.1. FASTRTPSGEN use....26
7.2. FASTRTPSGEN output....26
7.3. Executable use....26

1 Introduction

1.1 Purpose

Real-Time Publish-Subscribe (RTPS) is the wire interoperability protocol defined for the Data-Distribution Service (DDS) standard by the Object Management Group (OMG) consortium. This protocol standard is also defined by the OMG in the specification document “The Real-Time Publish-Subscribe Wire Protocol. DDS Interoperability Wire Protocol Specification (DDS-RTPS)” that can be obtained from the OMG web-page. The main objective of this specification is that all participants in a DDS communication structure, even if they use different vendor's implementations, can interoperate.

However, most existing RTPS implementations are either included in full DDS implementations (for example, RTI-DDS, OpenDDS,...) or use past versions of the specification (ORTE). For this reason, a standalone open access RTPS implementation will likely be welcome as a useful tool for developers worldwide.

1.2 Scope

The scope of this implementation is limited by the RTPS protocol specification of the OMG. Since the main purpose is to facilitate the implementation of a standalone RTPS wire protocol as close to the specification as possible, the included features are the ones described in the OMG document. The OMG specification defines message formats, interpretation and usage scenarios for any application willing to use RTPS protocol.

The most important features are:

- Enable best-effort and reliable publish-subscribe communications for real-time applications.
- Plug and play connectivity so that any new applications are automatically discovered by any other members of the network.
- Modularity and scalability to allow continuous growth with complex and simple devices in the network.

This implementation is coded using C++ as the programming language.

1.3 Intended Audience

The intended audience for this document are the software engineers willing to use *eProsima Fast RTPS* to develop their own RTPS applications. As explained above, this implementation uses C++, so the reader should have a basic understanding of this language. Some previous knowledge of RTPS will also be useful.

1.4 Reference Material

The reader should look into the OMG's RTPS specification document that can be found in the following web-page: http://www.omg.org/spec/DDSI-RTPS/2.2/

Additionally the user should also consult the eProsima FASTRTPSGEN User Manual as well as the Public API doxygen documentation available.

1.5 Document organization

This document can be divided into two main parts. The first part (composed of Chapters 2 and 3) contains a general view of the implementation specification. The second part describes the most important classes and methods and additional observations that need to be taken into account .

A brief summary of each chapter is presented in the following points:

- Chapter 2 (Implementation overview: scope and limitations): This chapter contains an overview of the implementation specification; as well as an introduction to the implementation scope and limitations.
- Chapter 3 (Implementation Specification): This chapter presents the implementation design. The four relevant modules are presented and explained.
- Chapter 4 (Object Specification): This chapter summarizes the classes and structures defined in this library.
- Chapter 5 (Publisher – Subscriber Interface Layer): This chapter explains the main classes of the Publisher-Subscriber layer and presents real examples of how they can be used.
- Chapter 6 (RTPS User interface): This chapter introduces the RTPS layer for advanced users that do not want to use the Publisher – Subscriber layer.
- Chapter 7 (Automatic code generation): This chapter has a brief introduction to the FASTRTPSGEN code generation tool. Please, read the FASTRTPSGEN user manual included in this distribution for more information.

2 Implementation overview: scope and limitations

There are multiple DDS and RTPS bundle implementations (e.g., RTI DDS[1], OpenDDS[2], OpenSpliceDDS[3], …) and some standalone RTPS implementations (e.g., ORTE[4]). However, this last group is somewhat reduced and the implementations they provided are antiquated (e.g., ORTE implements RTPS protocol 1.2 with a C and Java API). For this reason a new up-to-date accessible RTPS protocol implementation is being designed and released by eProsima.

2.1 Implementation scope

The *eProsima Fast RTPS* implementation follows the RTPS specification document as close as possible. This entails that, where possible, the same classes, attributes and methods names are used and implemented.

eProsima Fast RTPS is released as a standalone library with an API directly accessible by user applications.

2.2 Implementation limitations

Ref	Issue	Scope
1.	Not all QoS are supported in the implementation	Reliability, Durability and Liveliness Qos are embedded in the library behaviour. Any other Qos must be implemented by the user. Examples on how to implement certain Qos are included in the Publisher-Subscriber layer description.

2.3 Implementation Dependencies

List the main dependencies regarding the design effort.

Ref	Dependency	Action
1.	Boost libraries (1.57)	Boost asio will be used to manage sockets and communications; as well as other less important boost libraries (PropertyTree to read XML files, for example). The Boost libraries are needed to compile the library and to run some of the examples.

1 https://www.rti.com/products/dds/

2 http://www.opendds.org/

3 http://www.prismtech.com/opensplice

4 http://orte.sourceforge.net/

3 Implementation Specification

The eProsima implementation of the RTPS wire protocol will provide the users with the means to easily exchange RTPS messages in a RTPS network. This exchange will be managed through two different layers of user APIs. These APIs together with the other modules and dependencies will be explained in the following sections. A general view of the system architecture is also presented.

3.1 System Architecture

A general view of the eProsima Fast RTPS library is shown in Figure 3.1. The Fast RTPS library provides the user with two different layers to access its capabilities. The first is the RTPS layer, contained in the RTPSDomain. Using this layer the user can directly create RTPSWriter and RTPSReader and acces their corresponding Histories. To facilitate the use of this layer an additional Publication-Subscription layer has been developed. This second layer allows the user to create Publisher and Subscribers in certain topics and transmit user-defined TopicDataTypes between them.

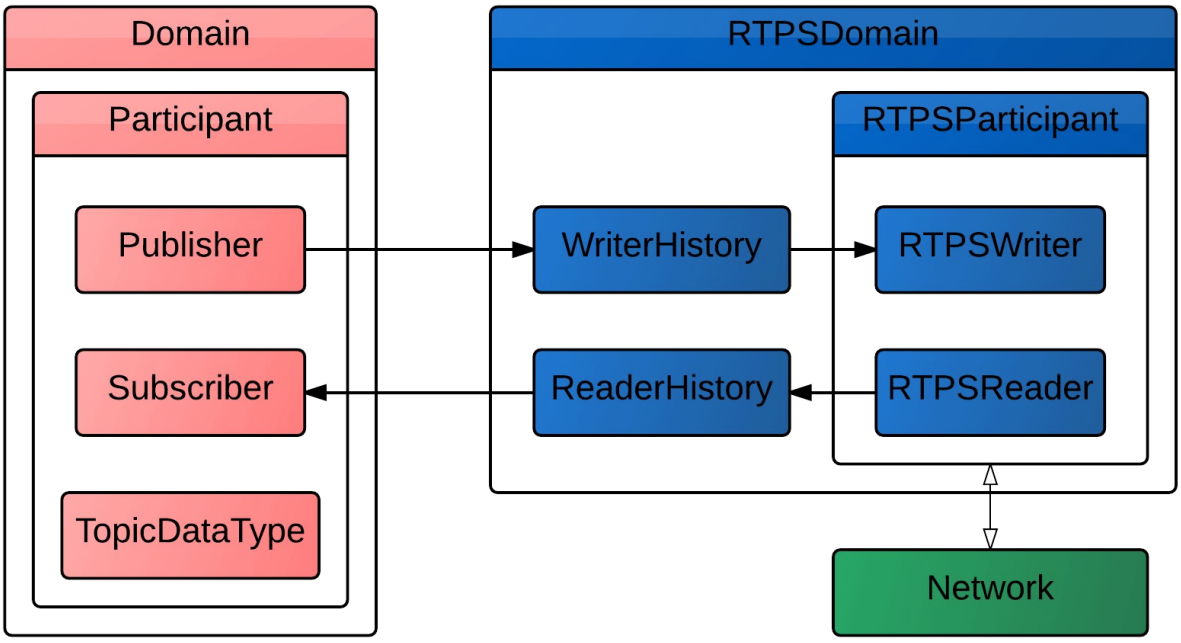

Figure 3.1: Fast RTPS general architecture

The different modules included in Figure 3.1 and their interaction with the user and between them will be explained in following sections.

3.2 Behavior Implementation

The correct behavior of the RTPS protocol will be achieved by using an event-based implementation. Multiple events will be handled by different threads allowing the application to efficiently tackle the different tasks. The thread structure, as well as the main events will be discussed in this section.

3.2.1 Thread structure

For each RTPS participant, different threads will be created that will manage different aspects of the RTPS implementation. Each application using this implementation will have at least these threads:

- **Main thread:** This thread is managed by the application or the user. The user interacts with the objects through the defined public API's.
- **Event thread(s):** This thread is in charge of processing the different events triggered wither by some periodic condition or by actions performed by the user in the main thread. In this version a single event thread is created and used per Participant.
- **Receive threads:** These threads are in charge of listening to the different ports. Since these threads are blocked until an RTPS message is received there would be a different thread for each different IP-port combination the Participant is listening from. Multiple endpoints can be assigned to the same listen thread.

3.2.2 Resource structure

There are three main types of resources in the implementation, that directly correspond with three classes. All resources are managed by the Participant.

- **ResourceListen**: Each listen resource is assigned to a single IP:port combination. It receives and process messages directed to that socket and performs the necessary actions in one or more of the associated Writers or Readers. In this version each ResourceListen runs a single thread listening to a single socket. Future versions may alter this behavior assigning multiple sockets (multiple ResourceListen objects) to a single listen thread.
- **ResourceEvent**: This resource manages the time-delayed event triggered periodically or by some message-based event. A single resource is implemented per participant, with a single thread performing all the actions. Future versions may include multiple ResourceEvents running in multiple threads to improve performance.
- **ResourceSend**: This resource manages all send operations in the Participant. This means that all endpoints included in a Participant send their messages through the socket defined in this resource. All messages are sent synchronously. Future versions will include multiple ResourceSend objects and the possibility to asynchronously send messages.

3.2.3 Main events

There are multiple events that are triggered wither directly by some action performed by the user, the reception of messages or even periodically. A list of the main events and the actions that need to be performed after them is included below, whereas a detailed description of all the events associated with each class of the design will be included in the detailed implementation chapter.

- **User-triggered events:** These events are triggered directly after the user performs some action, either directly to the RTPS Writer or its associated HistoryCache. These events are usually executed synchronously, directly in the Main thread, thus not using the ResourceEvent class.
- **Message-triggered events:** These events are triggered by the reception of an RTPS message. For example, the reception of an ACKNACK message would trigger a change in the status of some CacheChanges in the HistoryCache and, maybe, the re-send of some packets to a specific Reader.
- **Periodic events:** Some events must be periodically triggered according to RTPS rules. For example, heartbeat packages must be sent each heartbeatPeriod to all matching Readers.

4 Object Specification

Object oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem. It is one possible approach to software design. An object contains encapsulated data and procedures grouped together to represent an entity. The “object interface”, how the object can be interacted with, is also defined. An object-oriented program is described by the interaction of these objects.

This programming paradigm has been selected for this implementation since this library will use entities to represent each of the models integrating the protocol.

A table defining and explaining the most important object types is included in Table 1.

Object Name	Object Type	Comment
Publisher Subscriber Module		
Domain	Class	This class allows the user to control the creation and removal of participants, publishers and subscribers. It does not belong to the RTPS implementation but is included to manage the protocol.
ParticipantAttributes	Structure	This structure contains all the necessary parameters to create a Participant.
Participant	Class	This class contains a group of publishers and subscribers and registered TopicDataTypes.
ParticipantListener	Class	Base class to be used by the user to implement callbacks to certain actions.
TopicDataType	Class	This class is used to provide the Publishers and Subscribers with the methods to serialize, deserialize and get the key of a specific data type.
Publisher	Class	This class allows the user to send new information to the subscribers. Additionally, the class allows the user to add and remove Reader destinations where the messages need to be sent.
PublisherAttributes	Structure	This structure contains all the necessary parameters that define the behavior of a Publisher. It must be provided during construction.
PublisherListener	Class	Base class the user should use to implement specific callbacks to certain actions on the Publisher side.
Subscriber	Class	This class allows the user to retrieve information from the reader history. Additionally, the interface allows the user to add and remove Writer origins from where the messages are going to be sent if no Discovery Protocol is used.
SubscriberAttributes	Structure	This structure contains all the necessary parameters that define the behavior of a Subscriber. It must be provided during construction.

Object Name	Object Type	Comment
SubscriberListener	Class	Base class the user should use to implement specific callbacks to certain actions in the Subscriber side.
RTPSDomain		
RTPSParticipant	Class	Class that groups the created RTPSWriters and RTPSReaders.
RTPSParticipantAttributes	Structure	Necessary attributes to create an RTPSParticipant
RTPSDomain - Writer		
RTPSWriter	Class	This class implements the Writer side of the RTPS protocol allowing the user to send information to matched readers.
WriterListener	Class	Base class to implement callbacks to certain actions in the Writer side.
WriterAttributes	Structure	This structure defines the necessary attributes to create an RTPSWriter.
WriterHistory	Class	This class implements the Writer History. The user can add CacheChange_t to this history and the Writer will send it to the associated readers.
StatelessWriter	Class	This interface controls the *ReaderLocators* in Writer in case the user defined the RTPSWriter as BEST_EFFORT. This implementation is optimized for scalability. It Is suitable for best-effort communication, since it does not keep state information about remote entities.
ReaderLocator	Class	This type is used by the *StatelessWriter* to keep track of the location of its matching remote *Readers* and it manages the changes in WriterHistory as they relate to that particular Reader.
StatefulWriter	Class	This interface controls the *ReaderProxys* in the Writer in case the user defined the RTPSWriter as RELIABLE. It guarantees reliable communication and it is able to apply Qos-based or content-based filtering on the Writer side.
ReaderProxy	Class	This type represents the information the *Writer* maintains on each matched *RTPSReader* and manages the changes in *WriterHistory* as they relate to that particular Reader.
ChangeForReader	Structure	This object maintains information of a *CacheChange* in the *WriterHistory* as it pertains to the *Reader* represented by the *ReaderProxy*.
RTPSDomain - Reader		
RTPSReader	Class	This class implements the Reader side of the RTPS protocol allowing the user to receive information from matched writers.
ReadeListener	Class	Base class to implement callback to certain actions on the Reader side.
ReaderAttributes	Structure	This structure defines the necessary attributes to create a RTP-

Object Name	Object Type	Comment
		SReader.
ReaderHistory	Class	This class implements the Reader History. The RTPSReader adds CacheChanges to this History and the user can read them.
StatefulReader	Class	This interface controls the publishers that are matched with this Reader maintaining a state on each matched Writer. This state is encapsulated in the WriterProxy object.
WriterProxy	Class	This type represents the information the *StatefulReader* maintains on each matched writer.
ChangeFromWriter	Structure	This structure maintains information of a *CacheChange* in the *Reader HistoryCache* as it pertains to the *RTPSWriter* represented by the *WriterProxy*.
RTPSDomain - Builtin		
ResourceEvent	Class	Class used to manage the temporal events.
ResourceListen	Class	Class used to listen to a specific socket for RTPS messages.
ResourceSend	Class	Class used to manage the send operation.
TimedEvent	Class	Base class used to specify each of the temporal events.
PDPSimple	Class	This class announces the participant to predefined Locators in the network as well as to all other Participants that have been discovered.
EDPStatic	Class	This class reads and processes the XML file where the static Endpoints are described. It also performs endpoint matching.
EDPSimple	Class	This class implements the Simple Endpoint Discovery Protocol.
RTPSDomain - Common		
CacheChange_t	Structure	This structure is used to represent each change added to the *HistoryCache.*
Data	Data	This represents the data values associated with a particular change. Some changes may not have any associated data.
Message	Structure	Structure defining an RTPS Message. It is composed by a *Header* and one or multiple *Submessages.*
Header	Structure	This structure must appear at the beginning of each RTPS Message. It identifies the message as belonging to the RTPS protocol. It identifies the protocol version and the vendor that send the messages.
SubmessageHeader	Structure	This structure identifies the kind of *Submessage* and the optional elements within that *Submessage*.
MessageReceiver	Class	This object type interprets each of the received RTPS Messages. Since the meaning of each *Submessage* may depend on the previous *Submessages* a state of the whole *Message* must be maintained.

Object Name	Object Type	Comment
		This object type is reset each time a new *Message* is received.
RTPSMessageCreator	Class	This class allows the generation of serialized CDR RTPS Messages.
External Dependencies		
Boost ASIO	API	This API allows the control of the asynchronous events.
Boost Signals2	API	This API allows to add multiple tasks to a single event.

Table 1: Objects included in the implementation design

4.1 Data Design

This section describes the data design; i.e., the structures the software defines and uses. The data structures this library uses can be divided in three different groups:

- Internal Data Structures: Data exchanged among implementation modules.
- Global Data Structures: Data available to major portions of the architecture.
- Temporary Data Structures: Data created for temporal use.

These three groups are described in the following sections.

4.1.1 Internal Data Structures

The Internal Data Structures are passed among the architecture modules. In this case, the internal data structures are the *CacheChanges* stored in each *History*. Both the *RTPSWriter* and the *RTPSReader* use this type of structure to modify its own *History* and to create RTPS *Messages*.

4.1.2 Global Data Structures

The Global Data Structures are data that need to be known by major portions of the architecture.; i.e., data shared between the modules and with the end user. For our implementation these classes are: *ReaderLocator*, *ReaderProxy* and *WriterProxy*.

4.1.3 Temporary Data Structures

The most important Temporary Data Structures are the RTPS Messages. These structures are usually created from *CacheChanges* and eliminated after they are sent.

4.2 Common Data Types

The types of attributes that compose each of the classes of this implementation are summarized in Table 2. A detailed explanation of their purpose can be found in Table 8.2 of the OMG RTPS Specification document (OMG.RTPS.2.2 from now on).

Attribute type	Attribute Implementation	OMG.RTPS.2.2 Reference	
		Purpose	Mapping and Reserved values
GUID_t	typedef struct{ GuidPrefix_t guidPrefix; EntityId_t entityId; } GUID_t	Table 8.2	Page 153
GuidPrefix_t	typedef octet[12] GuidPrefix_t;	Table 8.2	Page 150
EntityId_t	typedef struct { octet[3] entityKey; octet entityKind; } EntityId_t;	Table 8.2	Pages 150-152
Locator_t	typedef struct{ long kind; unsigned long port; octet[16] address; } Locator_t	Table 8.2	Page 155
SequenceNumber_t	typedef struct{ long high; unsigned long low } SequenceNumber_t	Table 8.2	Page154
TopicKind_t	typedef struct{ long value; } TopicKind_t	Table 8.2	Page 155

ChangeKind_t	typedef unsigned long ChangeKind_t;
	const ChangeKind_t ALIVE = 0x0001 << 0;
	const ChangeKind_t NOT_ALIVE_DISPOSED = 0x0001 << 1;
	const ChangeKind_t NOT_ALIVE_UNREGISTERED = 0x0001 << 2;
		Table 8.2	Obtained from the DDS Specification Document.

InstanceHandle_t	typedef struct{ octet value[16]; }InstanceHandle_t;	Table 8.2	-
ProtocolVersion_t	typedef struct{ octet major; octet minor; } ProtocolVersion_t	Table 8.2	Page 156
VendorId_t	typedef struct{ octet[2] vendorId; } VendorId_t	Table 8.2	Page 153
Time_t	typedef struct{ long seconds; unsigned long fraction; } Time_t	Table 8.13	Page 153
Count_t	typedef struct {long value;} Count_t	Table 8.13	Page 156
StatusInfo_t	typedef struct {octet[4] value;} StatusInfo_t	Page 191	Page 156

Attribute type	Attribute Implementation	OMG.RTPS.2.2 Reference	
		Purpose	Mapping and Reserved values
Duration_t	typedef Time_t Duration_t;	Table 8.46	-
Delay_t	typedef Time_t Delay_t;	Table 8.46	-
ProtocolId_t	enum ProtocolId_t{ PROTOCOL_RTPS = 'RTPS'};	Table 8.13	

SampleInfo_t	class SampleInfo_t{
	ChangeKind_t sampleKind;
	GUID_t writerGUID;
	uint16_t ownershipStrength;
	Time_t sourceTimestamp;
	InstanceHandle_t iHandle;
	}
		-	-

MatchingStatus	Enum MatchingStatus {
	MATCHED_MATCHING,
	REMOVED_MATCHING
	}		

MatchingInfo	Class MatchingInfo {
	MatchignStatus status;
	GUID_t remoteEndpointGUID;
	}		

Table 2: Class Attributes Implementation

5 Publisher – Subscriber Interface Layer

This section explains the Publisher-Subscriber Layer. It presents the main classes and real examples of how they can be used. For a detailed explanation of each class and all its methods please consult the doxygen documentation.

5.1 Domain

This supervisor class allows the user to create Participants, Publishers and Subscribers. It also allows the user to register user-defined types. This class has only static methods.

5.2 Participant

The Participant is the grouping class in the publisher-subscriber layer. All Publisher, Subscribers and TopicDataTypes are stored in a participant. An example of the creation of a Participant can be observed here:

```
ParticipantAttributes PParam;
Pparam.rtps.setName("participant");
Pparam.rtps.builtin.domainId = 80;
Participant* p = Domain::createParticipant(PParam);
if(p!=nullptr){
//Participant correctly created
}
//To remove:
Domain::removeParticipant(p);
```

There are some parameters that must be defined:

- DomainId: This parameter is used to calculate the discovery ports and it is important to separate different applications working in the same network.
- Discovery: If the static endpoint discovery protocol is used then the filename must be provided.
- A maximum of 1000 Publishers and 1000 Subscribers can be created. Future versions will allow the user to determine this number per participant.

5.3 Type definition and registration

To register a type a class that inherits from TopicDataType must be defined. The user is responsible to implement the serialize and deserialize methods (mandatory) and the getKey method. eProsima Fast RTPS provides an automatic code generation tool that takes care of this definitions. This tool is briefly explained in Chapter 7 (Automatic code generation) and has its own User Manual.

An example of a user-defined type is shown below:

```
typedef struct TestType{
    char name[6]; //@key
    int32_t value;
    TestType()
    {
        value = -1;
        COPYSTR(name,"UNDEF"); //Your preferred string copy method
    }
    void print()
    {
        printf("Name: %s",name);
        cout << " |Value: "<< value << endl;
    }
}TestType;
class TestTypeDataType:public TopicDataType
{
public:
    TestTypeDataType()
{
        setName("TestType");
        m_typeSize = 6+4;
        m_isGetKeyDefined = true;
};
    ~TestTypeDataType(){};
    bool serialize(void*data,SerializedPayload_t* payload);
    bool deserialize(SerializedPayload_t* payload,void * data);
    bool getKey(void*data,InstanceHandle_t* ihandle);
};
```

An example implementation of these methods is included in below:

```
bool TestTypeDataType::serialize(void*data,SerializedPayload_t* payload){
    MyType* ptype = (MyType*)data;
    payload->length = 6+4;
     memcpy(payload->data,ptype->name,6);
     memcpy(payload->data+6,ptype->value,4);
    return true;
}
bool TestTypeDataType::deserialize(SerializedPayload_t* payload,void * data){
     MyType* ptype = (MyType*)data;
     memcpy(&ptype->name,payload->data,6);
     memcpy(&ptype->value,payload->data+6,4);
     return true;
}
bool TestTypeDataType::getKey(void*data,InstanceHandle_t* handle){
    TestType* tp = (TestType*)data;
    handle->value[0]  = 0;
    handle->value[1]  = 0;
    handle->value[2]  = 0;
    handle->value[3]  = 5; //length of string in CDR BE
    handle->value[4]  = tp->name[0];
    handle->value[5]  = tp->name[1];
    handle->value[6]  = tp->name[2];
    handle->value[7]  = tp->name[3];
    handle->value[8]  = tp->name[4];
    for(uint8_t i=9;i<16;i++)
        handle->value[i]  = 0;
    return true;
}
```

Finally, to register the type an instance of the defined class must be created and passed to the Domain through the provided interface, as shown here:

```
int main()
{
Participant* part; //CREATE A PARTICIPANT SOMEWHERE
...
TestTypeDataType TestTypeData;
Domain::registerType(part,(TopicDataType*)&TestTypeData);
...
};
```

Figure 5.1: Test Type registration

5.4 Publisher

There are different ways to create a publisher and an associated PublisherListener. One way is to create the publisher inside a class that inherits from PublisherListener. That way you can access the pointer to the publisher directly from the PublisherListeners callbacks.

5.4.1 Publisher-based class

```
class MyPublisher : public PublisherListener
{
	Participant* mp_participant;
	Publisher* mp_publisher;
	MyPublisher(){
		//Create participant(or receive it from somewhere else)
		//Type “TestType” must be registered (can be done outside the class).
		PublisherAttributes Wparam;
		Wparam.topic.topicKind = NO_KEY; //Other possible value: WITH_KEY
		Wparam.topic.topicDataType = "TestType";
		Wparam.topic.topicName = "Test_topic";
		//Modify the rest of the parameters as you wish
		mp_publisher =
			Domain::createPublisher(mp_participant,Wparam,this);
		if(mp_publisher == nullptr)
			cout << "Publisher creation failed" << endl;
	}
	~MyPublisher(){  Domain::removePublisher(mp_publisher);
	 }
	void onHistoryFull(Publisher* pub)	{
		mp_publisher->removeMinSeqCache();
	}
	void onPublicationMatched(Publisher* pub,MatchingInfo info)	{
		if(info.status == MATCHED_MATCHING)
			cout << “Publication Matched” << endl;
	}
	 void run(){
		if(mp_publisher!=nullptr){
			TestType obj;
			//Fill the structure
			mp_publisher->write(&obj);
		 }
	 }
};
```

The Domain performs a number of verifications before actually creating the Publisher:

- If the Participant was created using Static Endpoint Discovery Protocol, then the userDefinedId must be defined and greater than zero.
- The data type must have been registered before creating the Publisher.
- If the topic is defined as WITH_KEY, the registered data type must have its getKey method implemented.

In this example the creation of the Participant and the Publisher is performed in the constructor of the PublisherListener. This method provides the PublisherListener methods access to all the variables created during the construction. However, this is not mandatory. The creation could be performed elsewhere and then the pointer to be given as a argument to the class.

It is important to avoid time consuming loops or blocking sentences in the listener methods since that would cause the event and or listening threads to block.

5.5 Subscriber

5.5.1 Subscriber creation

A similar example to the Publisher definition is included below for a Subscriber.

```
class MySubscriber : public SubscriberListener
{
	Participant* mp_participant;
	Subscriber* mp_subscriber;
	MySubscriber(){
		//Create participant;
		//Type “TestType” must be registered (can be done outside the class).
		SubscriberAttributes Rparam;
		Rparam.topic.topicKind = NO_KEY; //Other possible value: NO_KEY
		Rparam.topic.topicDataType = "TestType";
		Rparam.topic.topicName = "Test_topic";
		//Modify the rest of the parameters as you wish
		mp_subscriber =
			Domain::createSubscriber(mp_participant,Rparam,this);
		if(mp_subscriber == nullptr)
			cout << "Subscriber creation failed" << endl;
	}
	~MySubscriber { Domain::removeSubscriber(mp_subscriber)  }
	void onSubscriptionMatched(Subscriber* sub,MatchingInfo info)	{
		if(info.status == MATCHED_MATCHING)
			cout << “Subscriber Matched” << endl;
	}
	void onNewDataMessage(Subscriber* sub) {
		TestType obj;
		SampleInfo_t info;
		if(mp_subscriber->takeNextData(&obj,&info)) {
			if(info.sampleKind == ALIVE)	{
				//print the data object fields
			}
		}
	}
	//If you read or take the data in the onNewDataMessage callback you should
	//avoid using waitForNewMessage() in you main loop because it would never exit.
	void run(){
		cout << "Enter number to exit: " << endl;
		int aux;
		cin >> aux;
	}
};
```

5.6 QoS Examples

In order to help with the implementation of additional Qos, an additional structure is provided each time a sample is read or taken from the history. This structure is of type SampleInfo_t and contains:

- sampleKind: Indicates whether the sample is ALIVE (a data structure of the type of the topic) or DISPOSED or UNREGISTERED.
- WriterGUID: GUID_t of the writer of the sample (useful to implement ownership Qos).
- OwnershipStrength: ownership strength of the writer when the sample was received.
- SourceTimestamp: timestamp indicating when the sample was send, useful to implement time-based filters.

Another important structure that helps with the implementation of additional QoS is the one provided with each call to the *onPublicationMatched* or *onSubscriptionMatched* method. This structure has the following parameters:

- remoteEndpointGUID: GUID_t of the writer or reader that has been matched.
- Status: MATCHED_MATCHING or REMOVED_MATCHING, indicating whether the method has been called when a new endpoint has been matched or when one of the already matched endpoints has been removed.

The next subsections include different examples of additional Qos that can be easily implemented with the library API.

5.6.1 Time-Based Filter and Content-Based Filter Qos

An example of this behavior implementation is shown below.

```
class MySubscriber: public SubscriberListener
{
	Subscriber* mp_sub;
	SubscriberAttributes m_att;
	SampleInfo_t m_lastInfo;
	void onNewDataMessage(Subscriber* sub)
	{
		TestType tp;
		SampleInfo_t info;
		if(mp_sub->readNextData((void*)&tp,&info)){ //Read Data
			if(info.sampleKind == ALIVE){
				if(passTimeFilter(info) && passContentFilter(tp)){
					m_lastInfo = info;
					//USE THE DATA
				}
			}
		}
	}
	bool passTimeFilter(SampleInfo_t& info)
	{
		if(TimeConv::Time_tAbsDiff2DoubleMillisec
			(m_lastInfo.sourceTimestamp,info.sourceTimestamp)
			>=
			TimeConv::Time_t2MilliSecondsDouble(
			m_att.qos.m_timeBasedFilter.minimum_separation))
		{
			return true;
		}
		return false;
	}
	bool passContentFilter(TesType& tp){
		//Check data against your filter and return true or false;
	}
}
```

5.6.2 Ownership Qos

If the ownership kind is set to EXCLUSIVE_OWNERSHIP_QOS, only one writer can update the sample sample. In this case the writer with the greatest ownership strength. An example of this behavior is included below. In this example we assume that only one key is used. For a multiple keyed topic, a vector of owners must be implemented since each key can have a different owner.

```
class MySubscriber : public SubscriberListener
{
TestType tp;
GUID_t ownerGUID;
uint32_t ownerStrength = 0;
bool hasOwner = false;
Subscriber* sub;
void MySubscriber::onNewDataMessage(Subscriber* sub) {
    if(sub->readNextData((void*)&tp,&info)){ //Read Data
    if(info.sampleKind == ALIVE)  {
        if(!hasOwner || info.ownershipStrength > ownerStrength ||
           (info.ownershipStrength == ownerStrength &&
                info.writerGUID < ownerGUID) //Change OWNER conditions
            )
        {
             hasOwner = true;
             ownerGUID = info.writerGUID;
             ownerStrength = info.ownershipStrength;
             sampleAccepted(tp);
        }
        else if(ownerGUID == writerGUID)
             sampleAccepted(tp);
    }
    }
}
void MySubscriber::sampleAccepted(TestType& tp) {
    //DO SOMETHING WITH THE SAMPLE
}
void MySubscriber::onSubscriptionMatched(Subscriber* sub,MatchingInfo info) {
    if(info.status ==MATCHED_MATCHING)   {
        cout << "Subscriber MATCHES Pub: " << info.remoteEndpointGuid << endl;
    }
    else if(info.status == REMOVED_MATCHING)   {
       cout << "Subscriber REMOVED Pub: " << info.remoteEndpointGuid << endl;
       if(info.remoteEndpointGUID == ownerGUID) {
            hasOwner = false;
       }
    }
}
}
```

6 RTPS User interface

Besides the provided publisher/subscriber interface layer, the user can directly create RTPSWriter and RTPSReaders and interact with eProsima Fast RTPS directly through the RTPS layer.

Each of the elements presented in the publisher-subscriber layer has an equivalent element in the RTPS layer. In fact, the provided publisher-subscriber layer has been created using the provided RTPS user available methods.

Nevertheless, advanced users may want to directly use the RTPS layer. For these users, an explanation of the RTPS user available API is included in the following sections.

6.1 RTPSDomain

The RTPSDomain is the supervisor class that allows the user to create and remove RTPSParticipant, RTPSWriter and RTPSReader. It has only static methods.

6.2 RTPSParticipant

This class is used to group multiple RTPSWriters and RTPSReaders into a single organization unit. It also manages the builtin discovery and liveliness protocols.

An example of the creation of an RTPSParticipant is shown below:

```
RTPSParticipantAttributes PParam;
Pparam.setName("participant");
Pparam.builtin.domainId = 80;
Pparam.use_IP6_to_send = false;
RTPSParticipant* p = RTPSDomain::createRTPSParticipant(PParam);
if(p!=nullptr){
//Participant correctly created
}
//To remove:
RTPSDomain::removeParticipant(p);
```

6.2.1 Endpoint registration

As mentioned before, the RTPSParticipant is also used to manage the builtin discovery protocols. The user needs to register each created RTPSWriter or RTPSReader in the builtin protocols manually. An example of this registration process is shown below:

```
RTPSParticipant* part;
RTPSWriter* writer;
TopicAttributes tA("topicName","topicType",NO_KEY);
WriterQos wqos; //Change QOS as you want
part->registerWriter(writer,tA,wqos);
...
part->updateWriter(writer,wqos);
```

It could be argued that the system should register each RTPSWriter and RTPSReader after they are created. However, to create a RTPS endpoint you don't need to define any QOS or topic attributes, and an automatic registration would force all users to provide those parameters upon creation. This way you can use RTPS endpoints with a minimum configuration, as shown in the next sections, and only define their QoS in case you want to register them.

6.3 RTPSWriter – WriterHistory

The RTPSWriter is the class in charge of sending the data to the matched readers. The user interacts with the RTPSWriter and with its associated WriterHistory. An example of the creation of the RTPSWriter-WriterHistory pair is shown below:

```
HistoryAttributes hatt;
WriterHistory * history = new WriterHistory(hatt);
WriterAttributes watt;
RTPSWriter* writer = RTPSDomain::createRTPSWriter(rtpsParticipant,watt,hist);
```

6.3.1 WriterAttributes

There are multiple parameters that can be set to define the behaviour of a Writer. For a complete list and a detailed explanation of each one please consult the doxygen documentation. The most important ones are described below:

- WriterAttributes.endpoint.reliabilityKind: This parameter can be set to RELIABLE (default) or BEST_EFFORT. It defines how the RTPSWriter relates to the matched readers. Two different specifications of the RTPSWriter class are used internally to manage this two different kinds of writers (StatefulWriter and StatelessWriter).
- WriterAttributes.endpoint.topicKind: This parameter can be set to NO_KEY (default) and WITH_KEY. For WITH_KEY writers the user should always provide the KEY of the CacheChange_t before adding it to the WriterHistory.
- WriterAttributes.endpoint.LocatorLists: There are two locator lists associated with an RTPSWriter (unicast and multicast). Also, this lists are only used for RELIABLE writers, since BEST_EFFORT writers don't need to receive messages.
- WriterAttributes.times: This structure define some periodic intervals regarding the behaviour of a RELIABLE writer with respect to certain events. For example, the user can define the ammount of time the writer should wait between sending Heartbeats.

6.3.2 Send Data with a RTPSWriter

To send Data with a RTPSWriter you first should request a CacheChange_t from the RTPSWriter and then add it to the corresponding WriterHistory. An example of this procedure is shown below, where a string is being send.

```
CacheChange_t* ch = writer->newCacheChange(ALIVE);
ch->serializedPayload->length = sprintf(ch->serializedPayload->data,"My String %d",2);
//Define ch->instanceHandle if the Writer is WITH_KEY
history->add_change(ch);
```

6.3.3 Matched readers

There is certain information an RTPSWriter needs to know about its matched readers. This information is contained in the RemoteReaderAttributes structure. The user can manually add matched readers to its writer. Of course, if the user registers the writer in the participant and the Endpoint Discovery Protocol is activated, the participant will automatically add the readers that match the Writer topic attributes and QoS. An example of a manual addition of a matched reader is shown below:

```
RemoteReaderAttributes ratt;
Locator_t loc;
loc.set_IP4_address(127,0,0,1);
loc.port = 22222;
ratt.endpoint.unicastLocatorList.push_back(loc)
ratt.guid = c_Guid_Unknown; //For Realiable Writers, you actually need the GUID_t
writer->matched_writer_add(ratt);
```

6.4 RTPSReader – ReaderHistory

The RTPSReader is in charge or receiving the data send by the writer. The user can interact directly with the RTPSReader and with the ReaderHistory. An example of the creation of a RTPSReader is shown below:

```
class MyReaderListener:public ReaderListener;
MyReaderListener listen;
HistoryAttributes hatt;
ReaderHistory * history = new ReaderHistory(hatt);
ReaderAttributes ratt;
RTPSReader* reader = RTPSDomain::createRTPSReader(rtpsParticipant,watt,hist,&listen);
```

6.4.1 ReaderAttributes

The RTPSReader attributes are almost identical to the WriterAttributes. The default value for the reliabilityKind in the RTPSReader is BEST_EFFORT. There are also two different implementations of the RTPSReader depending on the reliability kind (StatelessReader and StatefulReader).

Since all RTPSReaders need at least a Locator, if the two locator lists are left empty the system will assign the RTPSReader to the default participant locator.

6.4.2 Reading data from a RTPSReader

To read a CacheChange from the RTPSReader you can directly ask the reader for the next unread or untaken CacheChange, as shown here:

```
reader->waitForUnreadMessage(); //This method blocks the execution
CacheChange_t* change;
if(reader->nextUnreadCache(&change))
//DO SOMETHING WITH THE DATA
//LATER YOU CAN REMOVE IT FROM THE HISTORY
history->remove_change(change);
```

6.4.3 Receving data with a ReaderListener

There is a second possibility to receive data from a RTPSReader, and it is to use a ReaderListener. You can specify some callback methods by creating a class that inherits from ReaderListener. An example is shown below:

```
class MyReaderListener: public ReaderListener{
public:
MyReaderListener(){}
~MyReaderListener(){}
void onNewCacheChangeAdded(RTPSReader* reader,const CacheChange_t* const change){
    printf("%s\n",change->serializedPayload.data);
    reader->getHistory()->remove_change((CacheChange_t*)change);
}
}
```

7 Automatic code generation

An automatic code generation tool is included with the RTPS installation. This tool is a java *.jar* package that takes a type defined in an IDL file an generates the necessary code to create a publisher and a subscriber of that type. It can also generate a Makefile to compile the code in Linux or a Visual Studio project to compile the code in Windows. A brief introduction to this tool and the code it generates is included in the following sections. For more information please consult the FASTRTPSGEN manual included with this Fast RTPS installation.

7.1 FASTRTPSGEN use

Two scripts are included in the Fast RTPS installation to call the .jar, one for linux and one for windows. A possible call to this script would be:

```
fastrtpsgen.bat -d outputdir -example x64Win64VS2010 -replace MyType.idl
```

where MyType.idl would be a file containing our type definition, for example:

```
struct MyStruct{
     long id;
     string message;
};
```

7.2 FASTRTPSGEN output

The fastrtpsgen utility generates different files. The most important ones are described below:

- MyType.cxx/.h: These files contain the type definition.
- MyTypePublisher.cxx/.h: These files define the publisher with all the necessary options as well as a PublisherListener. The user must fill the needed methods for his application.
- MyTypeSubscriber.cxx/.h: These files contain the definition of the subscriber and the SubscriberListener. The behavior of the subscriber can be altered changing the methods implemented on these files.
- MyTypePubSubType.cxx/.h: These files contain the serialization and deserialization code for the type defined by the user. It also defines the getKey method.
- MyTypePubSubMain.cxx: Main file of the aplication. When the executable is created it accepts only a parameter, either subscriber or publisher indicating which side you want to execute.
- Makefiles or Visual studio project files.

7.3 Executable use

When the code compiles correctly the user can execute his application in two separate command prompts calling either *MyType.exe publisher* or *MyType.exe subscriber.*